PANDAS

DATA STRUCTURES AND DATA MANIPULATION TOOLS

NOR HAMIZAH MISWAN

GETTING STARTED

- Pandas contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python.
- Pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib.
- The biggest difference of NumPy and Pandas:
  - ➢Pandas is designed for working with tabular or heterogeneous data.
  - ➢NumPy is best suited for working with homogeneous numerical array data.

GETTING STARTED

- To import pandas: `import pandas as pd`
- It easier to import Series and DataFrame into the local namespace since they are so frequently used: `from pandas import Series, DataFrame`
- Series:
  - A one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index
- DataFrame:
  - represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.)..

SERIES

- A one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index.
- The simplest Series is formed from only an array of data: `obj = pd.Series([4, 7, -5, 3]).`
- We can create a Series with an index identifying each data point with a label:

```
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
obj2['a']
```

- To replace certain number: `obj2['d'] = 6`

```
obj2[['c', 'a', 'd']]
```

SERIES

- Can also used with NumPy functions or NumPy-like operations, such as scalar multiplication, or applying math functions:

```
obj2[obj2 > 0]
obj2 * 2
np.exp(obj2)
```

- If you have data contained in a Python dictionary, you can create a Series from it by passing the dictionary:

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = pd.Series(sdata)
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = pd.Series(sdata, index=states)
```

SERIES

- To detect missing data, use isnull or notnull functions:

```
pd.isnull(obj4)  #silimar to obj4.isnull()
pd.notnull(obj4)
```

- A Series’s index can be altered in-place by assignment:

```
obj = pd.Series([4, 7, -5, 3])
obj
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

DATAFRAME

- DataFrame,
  - represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.).
  - has both a row and column index; it can be thought of as a dictionary of Series all sharing the same index. ate arrays of 0s or 1s.
- To construct DataFrame from a dict of equal-length lists or NumPy arrays:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
frame.head() #specify the header (top five)
```

DATAFRAME

- To specify a sequence of columns: `pd.DataFrame(data, columns=['year', 'state', 'pop'])`
- If you pass a column that isn't contained in the dict, it will appear with missing values in the result:

```
frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                                          index=['one', 'two', 'three', 'four', 'five', 'six'])
```

- A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute: `frame2['state']`
  `frame2.year`
- To retrieved specific row by position or name using loc function: `frame2.loc['three']`
- To insert/change the values in a column: `frame2['debt'] = 16.5`
  `frame2['eastern'] = frame2.state == 'Ohio'`

DATAFRAME

- The del method can then be used to remove this column: `del frame2['eastern']`
- To transpose the DataFrame:

```
pop = {'Nevada': {2001: 2.4, 2002: 2.9}, 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
frame3 = pd.DataFrame(pop)
frame3.T
```

INDEXING AND REINDEXING

- An important method on pandas objects is reindex, which means to create a new object with the data conformed to a new index:

```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
obj3.reindex(range(6), method='ffill') #ffill refered to forward-fills the values.
```

INDEXING AND REINDEXING

- With DataFrame, reindex can alter either the (row) index, columns, or both. When passed only a sequence, it reindexes the rows in the result:

```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
        columns=['Ohio', 'Texas', 'California']
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
states = ['Texas', 'Utah', 'California']
frame.reindex(columns=states) #The columns can be reindexed with the columns keyword
```

- You can reindex by label indexing with loc function:

```
frame.loc[['a', 'b', 'c', 'd'], states]
```

DROP ENTRIES

- Dropping one or more entries from an axis is easy if you already have an index array or list without those entries.

```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
new_obj = obj.drop('c')
obj.drop(['d', 'c'])
data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio', 'Colorado', 'Utah', 'New
        York'], columns=['one', 'two', 'three', 'four'])
data.drop(['Colorado', 'Ohio'])
data.drop('two', axis=1) #drop values from the columns by passing axis=1 or axis='columns'
```

INDEXING, SLICING, FILTERING

- Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive.

```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
obj['b'] #same as obj[1]
obj[2:4]
obj[['b', 'a', 'd']]
obj[[1, 3]]
obj['b':'c']
obj['b':'c'] = 5 #modifies the corresponding value
```

INDEXING, SLICING, FILTERING

- Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio', 'Colorado', 'Utah', 'New York'], columns=['one', 'two', 'three', 'four'])
data['two']
data[['three', 'one']]
data[:2] #this will select based on row
```

INDEXING, SLICING, FILTERING

- Selection with loc and iloc,
  - ➢enable you to select a subset of the rows and columns from a DataFrame with NumPy-like notation using either axis labels (loc) or integers (iloc).

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio', 'Colorado', 'Utah',
'New York'], columns=['one', 'two', 'three', 'four'])
data.loc['Colorado', ['two', 'three']]
data.iloc[2, [3, 0, 1]]
data.iloc[2]
data.iloc[[1, 2], [3, 0, 1]]
data.loc[:'Utah', 'two']
```

ARITHMETIC METHOD

- An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes.
- When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs.

```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
s1 + s2
```

ARITHMETIC METHOD

- In the case of DataFrame, alignment is performed on both the rows and the columns:

```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'), index=['Ohio',
        'Texas', 'Colorado'])
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), index=['Utah',
        'Ohio', 'Texas', 'Oregon'])
df1 + df2
```

- Since the 'c' and 'e' columns are not found in both DataFrame objects, they appear as all missing in the result.
- Also, try:

```
1 / df1 #Table 5-5 shows the flexible arithmetic methods
```

SORTING

- Sorting a dataset by some criterion is another important built-in operation.
- To sort lexicographically by row or column index, use the sort_index method, which returns a new, sorted object:

```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
obj.sort_index()
```

- With a DataFrame, you can sort by index on either axis:

```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'], columns=['d', 'a',
        'b', 'c'])
frame.sort_index() #by default, it will sort in row
frame.sort_index(axis=1) #axis=1will sort by column
frame.sort_index(axis=1, ascending=False) #n be sorted in descending order
```

SORTING

- To sort a Series by its values, use its sort_values method:

```
obj = pd.Series([4, 7, -3, 2])
obj.sort_values()
```

- Any missing values are sorted to the end of the Series by default:

```
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
obj.sort_values()
```

- When sorting a DataFrame, you can use the data in one or more columns as the sort keys. To do so, pass one or more column names to the by option of sort_values:

```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame.sort_values(by='b')
frame.sort_values(by=['a', 'b']) #rank a first, then b
```

DESCRIPTIVE STATISTICS

- Pandas objects are equipped with a set of common mathematical and statistical methods.
- Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data.

```
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5], [np.nan, np.nan], [0.75, -1.3]], index=['a', 'b', 'c',
        'd'], columns=['one', 'two'])
df.sum()
df.sum(axis='columns') #or axis=1 add the column for each row
df.mean(axis='columns', skipna=False) #skip adding with NA using skipna function
```

Note: refer Table 5-8 for list of descriptive statistics